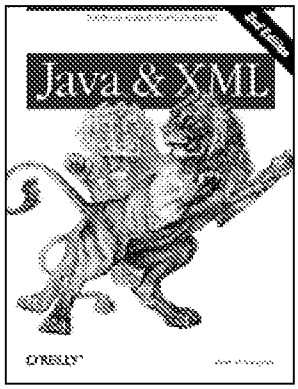


Java & XML, 2nd Edition

Solutions to Real-World Problems

By Brett McLaughlin
2nd Edition September 2001
0-596-00197-5, Order Number: 1975
528 pages, $44.95

Chapter 12
SOAP

In this chapter:

SOAP is the Simple Object Access Protocol. If you haven't heard of it by now, you've probably been living under a rock somewhere. It's become the newest craze in web programming, and is integral to the web services fanaticism that has taken hold of the latest generation of web development. If you've heard of .NET from Microsoft or the peer-to-peer "revolution," then you've heard about technologies that rely on SOAP (even if you don't know it). There's not one but *two* SOAP implementations going on over at Apache, and Microsoft has hundreds of pages on their MSDN web site devoted to it (http://msdn.microsoft.com).

In this chapter, I explain what SOAP is, and why it is such an important part of where the web development paradigm is moving. That will help you get the fundamentals down, and prepare you for actually working with a SOAP toolkit. From there, I briefly run over the SOAP projects currently available, and then delve into the Apache implementation. This chapter is not meant to be the complete picture on SOAP; the next chapter fills in lots of gaps. Take this as the first part of a miniseries; many of your questions at the end of this chapter will be answered in the next.

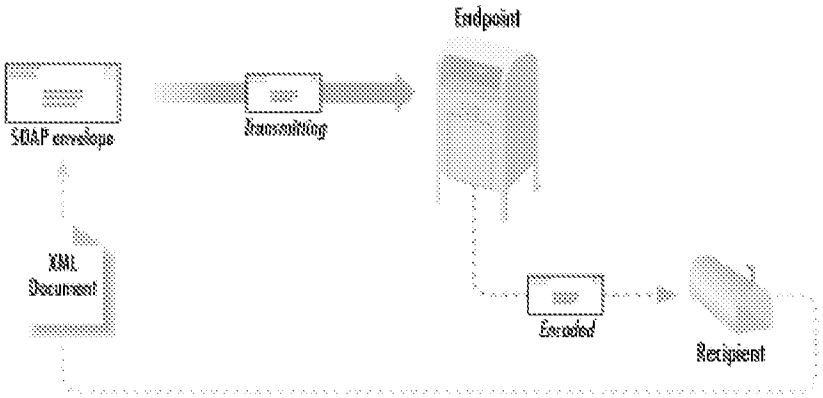
Starting Out

The first thing to do is get an understanding of what SOAP is. You can read through the complete W3C note submission, which is fairly lengthy, at http://www.w3.org/TR/SOAP. When you take away all of the hype, SOAP is just a protocol. It's a simple protocol (to use, not necessarily to write), based on the idea that at some point in a distributed architecture, you'll need to exchange information. Additionally, in a system that is probably overtaxed

and process-heavy, this protocol is lightweight, requiring a minimal amount of overhead. Finally, it allows all this to occur over HTTP, which allows you to get around tricky issues like firewalls and keep away from having all sorts of sockets listening on oddly numbered ports. Once you get that down, everything else is just details.

Of course, I'm sure you're here for the details, so I won't leave them out. There are three basic components to the SOAP specification: the SOAP envelope, a set of encoding rules, and a means of interaction between request and response. Begin to think about a SOAP message as an actual letter; you know, those antiquated things in envelopes with postage and an address scrawled across the front? That analogy helps SOAP concepts like "envelope" make a lot more sense. Figure 12-1 seeks to illustrate the SOAP process in terms of this analog.

Figure 12-1. The SOAP message process



With this picture in your head, let's look at the three components of the SOAP specification. I cover each briefly and provide examples that illustrate these concepts more completely. Additionally, it's these three key components that make SOAP so important and valuable. Error handling, support for a variety of encodings, serialization of custom parameters, and the fact that SOAP runs over HTTP make it more attractive in many cases than the other choices for a distributed protocol.[1] Additionally, SOAP provides a high degree of interoperability with other applications, which I delve into more completely in the next chapter. For now, I want to focus on the basic pieces of SOAP.

The Envelope

The SOAP envelope is analogous to the envelope of an actual letter. It supplies information about the message that is being encoded in a SOAP payload, including data relating to the recipient and sender, as well as details about the message itself. For example, the header of the SOAP envelope can specify exactly how a message must be processed. Before an application goes forward with processing a message, the application can determine information about a message, including whether it will even be able to process the message. Distinct from the situation with standard XML-RPC calls (remember that? XML-RPC messages, encoding, and the rest are all wrapped into a single XML fragment), with SOAP actual interpretation occurs in order to determine something about the message. A typical SOAP message can also include the encoding style, which assists the recipient in interpreting the message. Example 12-1 shows the SOAP envelope, complete with the specified encoding.

Example 12-1: The SOAP envelope

```
<soap:Envelope
   xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
   soap:encodingStyle="http://myHost.com/encodings/secureEncoding"
>
  <soap:Body>
    <article xmlns="http://www.ibm.com/developer">
      <name>Soapbox</name>
      <url>
        http://www-106.ibm.com/developerworks/library/x-soapbx1.html
      </url>
    </article>
  </soap:Body>
</soap:Envelope>
```

You can see that an encoding is specified within the envelope, allowing an application to determine (using the value of the `encodingStyle` attribute) whether it can read the incoming message situated within the `Body` element. Be sure to get the SOAP envelope namespace correct, or SOAP servers that receive your message will trigger version mismatch errors, and you won't be able to interoperate with them.

Encoding

The second major element that SOAP brings to the table is a simple means of encoding user-defined datatypes. In RPC (and XML-RPC), encoding can only occur for a predefined set of datatypes: those that are supported by whatever XML-RPC toolkit you download. Encoding other types requires modifying the actual RPC server and clients themselves. With SOAP, however, XML schemas can be used to easily specify new datatypes (using the `complexType` structure discussed way back in Chapter 2), and those new types can be easily represented in XML as part of a SOAP payload. Because of this integration with XML Schema, you can encode any datatype in a SOAP message that you can logically describe in an XML schema.

Invocation

The best way to understand how SOAP invocation works is to compare it with something you already know, such as XML-RPC. If you recall, an XML-RPC call would look something like the code fragment shown in Example 12-2.

Example 12-2: Invocation in XML-RPC

```
// Specify the XML parser to use
XmlRpc.setDriver("org.apache.xerces.parsers.SAXParser");

// Specify the server to connect to
XmlRpcClient client =
    new XmlRpcClient("http://rpc.middleearth.com");

// Create the parameters
Vector params = new Vector(  );
params.addElement(flightNumber);
params.addElement(numSeats);
params.addElement(creditCardType);
```

```
params.addElement(creditCardNum);

// Request reservation
Boolean boughtTickets =
    (Boolean)client.execute("ticketCounter.buyTickets", params);

// Deal with the response
```

I've coded up a simple ticket counter-style application. Now, look at Example 12-3, which shows the same call in SOAP.

Example 12-3: Invocation in SOAP

```
// Create the parameters
Vector params = new Vector(  );
params.addElement(
    new Parameter("flightNumber", Integer.class, flightNumber, null));
params.addElement(
    new Parameter("numSeats", Integer.class, numSeats, null));
params.addElement(
    new Parameter("creditCardType", String.class, creditCardType, null));
params.addElement(
    new Parameter("creditCardNumber", Long.class, creditCardNum, null));

// Create the Call object
Call call = new Call(  );
call.setTargetObjectURI("urn:xmltoday-airline-tickets");
call.setMethodName("buyTickets");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
call.setParams(params);

// Invoke
Response res = call.invoke(new URL("http://rpc.middleearth.com"), "");

// Deal with the response
```

As you can see, the actual invocation itself, represented by the `Call` object, is resident in memory. It allows you to set the target of the call, the method to invoke, the encoding style, the parameters, and more not shown here. It is more flexible than the XML-RPC methodology, allowing you to explicitly set the various parameters that are determined implicitly in XML-RPC. You'll see quite a bit more about this invocation process in the rest of the chapter, including how SOAP provides fault responses, an error hierarchy, and of course the returned results from the call.

With that brief introduction, you probably know enough to want to get on with the fun stuff. Let me show you the SOAP implementation I'm going to use, explain why I made that choice, and get to some code.

Setting Up

Now that you have some basic concepts down, it's time to get going on the fun part, the code. You need a project or product for use, which turns out to be simpler to find than you might think. If you want a Java-based project providing SOAP capability, you don't have to look that far. There are two groups of products out there: commercial and free. As in most

of the rest of the book, I'm steering away from covering commercial products. This isn't because they are bad (on the contrary, some are wonderful); it's because I want every reader of this book to be able to use every example. That calls for accessibility, something commercial products don't provide; you have to pay to use them, or download them and at some point the trial period runs out.

That brings us to open source projects. In that realm, I see only one available: Apache SOAP. Located online at http://xml.apache.org/soap, this project seeks to provide a SOAP toolkit in Java. Currently in a Version 2.2 release, you can download it from the Apache web site. That's the version and project I use for the examples throughout this chapter.

Other Options

Before moving on to the installation and setup of Apache SOAP, I will answer a few questions that might be rattling around in your head. It's probably clear why I'm not using a commercial product. However, you may be thinking of a couple of other open source or related options that you might want to use, and wondering why I am not covering those.

What about IBM SOAP4J?

First on the list of options is IBM's SOAP implementation, IBM SOAP4J. IBM's work is actually the basis of the current Apache SOAP project, much as IBM XML4J fed into what is now the Apache Xerces XML parser project. Expect the IBM implementation to resurface, wrapping the Apache SOAP project's implementation. This is similar to what is happening with IBM's XML4J; it currently just provides IBM packaging over Xerces. This makes some additional levels of vendor-backing available to the open source version, although the two (Apache and IBM) projects are using the same codebase.

Isn't Microsoft a player?

Yes. Without a doubt, Microsoft and its SOAP implementation, as well as the whole .NET initiative (covered more in the next chapter), are very important. In fact, I wanted to spend some time covering Microsoft's SOAP implementation in detail, but it only supports COM objects and the like, without Java support. For this reason, coverage of it doesn't belong in a book on Java and XML. However, Microsoft (despite the connotations we developers tend to have about the company) is doing important work in web services, and you'd be making a mistake in writing it off, at least in this particular regard. If you need to communicate with COM or Visual Basic components, I highly recommend checking out the Microsoft SOAP toolkit, found online at http://msdn.microsoft.com/library/default.asp?url=/nhp/Default.asp?contentid=28000523 along with a lot of other SOAP resources.

What's Axis?

Those of you who monitor activity in Apache may have heard of Apache Axis. Axis is the next-generation SOAP toolkit, also being developed under the Apache XML umbrella. With SOAP (the specification, not a specific implementation) undergoing fairly fast and radical change these days, tracking it is difficult. Trying to build a version of SOAP that meets current requirements and moves with new development is also awfully tough. As a result, the current Apache SOAP offering is somewhat limited in its construction. Rather than try to rearchitect an existing toolkit, the Apache folks started fresh with a new codebase and

project; thus, Axis was born. Additionally, the naming of SOAP was apparently going to change, from SOAP to XP and then to XMLP. As a result, the name of this new SOAP project was uncoupled from the specification name; thus, you have "Axis." Of course, now it looks like the W3C is going back to calling the specification SOAP (Version 1.2, or Version 2.0), so things are even more confusing!

Think of IBM SOAP4J as architecture 1 of the SOAP toolkit. Following that is Apache SOAP (covered in this chapter), which is architecture 2. Finally, Axis provides a next-generation architecture, architecture 3. This project is driven by SAX, while Apache SOAP is based upon DOM. Additionally, Axis provides a more user-friendly approach in header interaction, something missing in Apache SOAP. With all of these improvements, you're probably wondering why I'm not covering Axis. It's simply too early. Axis is presently trying to get together a 0.51 release. It's not a beta, or even an alpha, really; it's very early on. While I'd love to cover all the new Axis features, there's no way your boss is going to let you put in a pre-alpha release of open source software in your mission-critical systems, now is there? As a result, I've chosen to focus on something you *can* use, *today*: Apache SOAP. I'm sure when Axis does finalize, I'll update this chapter in a subsequent revision of the book. Until then, let's focus on a solution you can use.

Installation

There are two forms of installation with regard to SOAP. The first is running a SOAP client, using the SOAP API to communicate with a server that can receive SOAP messages. The second is running a SOAP server, which can receive messages from a SOAP client. I cover installation of both cases in this section.

The client

To use SOAP on a client, you first need to download Apache SOAP, available online at http://xml.apache.org/dist/soap. I've downloaded Version 2.2, in the binary format (in the *version-2.2* subdirectory). You should then extract the contents of the archive into a directory on your machine; my installation is in the *javaxml2* directory (*c:\javaxml2* on my Windows machine, */javaxml2* on my Mac OS X machine). The result is */javaxml2/soap-2_2*. You'll also need to download the JavaMail package, available from Sun at http://java.sun.com/products/javamail/. This is for the SMTP transfer protocol support included in Apache SOAP. Then, download the JavaBeans Activation Framework (JAF), also from Sun, available online at http://java.sun.com/products/beans/glasgow/jaf.html. I'm assuming that you still have Xerces or another XML parser available for use.

> NOTE: Ensure your XML parser is JAXP-compatible and namespace-aware. Your parser, unless it's a very special case, probably meets both of these requirements. If you have problems, go back to using Xerces.
>
> NOTE: Use a recent version of Xerces; Version 1.4 or greater should suffice. There are a number of issues with SOAP and Xerces 1.3(.1), so I'd avoid that combination like the plague.

Expand both the JavaMail and JAF packages, and then add the included *jar* files to your classpath, as well as the *soap.jar* library. Each of these *jar* files is either in the root directory or in the *lib/* directory of the relevant installation. At the end, your classpath should look

something like this:

```
$ echo $CLASSPATH
/javaxml2/soap-2_2/lib/soap.jar:/javaxml2/lib/xerces.jar:
/javaxml2/javamail-1.2/mail.jar:/javaxml2/jaf-1.0.1/activation.jar
```

On Windows, it should look like:

```
c:\>echo %CLASSPATH%
c:\javaxml2\soap-2_2\lib\soap.jar;c:\javaxml2\lib\xerces.jar;
c:\javaxml2\javamail-1.2\mail.jar;c:\javaxml2\jaf-1.0.1\activation.jar
```

Finally, add the *javaxml2/soap-2_2/* directory to your classpath if you want to run the SOAP examples. I cover setup for specific examples in this chapter as I get to them.

The server

To build a SOAP-capable set of server components, you first need a servlet engine. As in earlier chapters, I'll use Apache Tomcat (available from http://jakarta.apache.org) throughout this chapter for examples. You'll then need to add everything needed on the client to the server's classpath. The easiest way to do that is to drop *soap.jar*, *activation.jar*, and *mail.jar*, as well as your parser, in your servlet engine's library directory. On Tomcat, this is simply the *lib/* directory, which contains libraries that should be autoloaded. If you want to support scripting (which is not covered in this chapter, but is in the Apache SOAP examples), you'll need to put *bsf.jar* (available online at http://oss.software.ibm.com/developerworks/projects/bsf) and *js.jar* (available from http://www.mozilla.org/rhino/) in the same directory.

> NOTE: If you are using Xerces with Tomcat, you'll need to perform the same renaming trick I talked about in Chapter 10. Rename *parser.jar* to *z_parser.jar* and *jaxp.jar* to *z_jaxp.jar*, to ensure that *xerces.jar* and the included version of JAXP are loaded prior to any other parser or JAXP implementation.

Now restart your servlet engine, and you're ready to write SOAP server components.

The router servlet and admin client

In addition to basic operation, Apache SOAP includes a router servlet as well as an admin client; even if you don't want to use these, I recommend you install them so you can test your SOAP installation. This process is servlet-engine-specific, so I just cover the Tomcat installation here. However, installation instructions for several other servlet engines are available online at http://xml.apache.org/soap/docs/index.html.

Installation under Tomcat is simple; just take the *soap.war* file in the *soap-2_2/webapps* directory, and drop it in your *$TOMCAT_HOME/webapps* directory. That's it! To test the installation, point your web browser to *http://localhost:8080/soap/servlet/rpcrouter*. You should get a response like that shown in Figure 12-2.

Figure 12-2. The RPC router servlet

Although this looks like an error, it does indicate that things are working correctly. You should get the same response pointing your browser to the admin client, at *http://localhost:8080/soap/servlet/messagerouter*.

As a final test of both the server and client, ensure you have followed all the setup instructions so far. Then execute the following Java class as shown, supplying your servlet URL for the RPC router servlet:

```
C:\>java org.apache.soap.server.ServiceManagerClient
        http://localhost:8080/soap/servlet/rpcrouter list
Deployed Services:
```

You should get the empty list of services, as shown here. If you get any other message, consult the long list of possible errors at http://xml.apache.org/soap/docs/trouble/index.html. A fairly complete list of problems that you can run into is there. If you do get the empty list of services, then you're set up and ready to continue with the examples in the rest of this chapter.

Getting Dirty

There are three basic steps in writing any SOAP-based system, and I'll look at each in turn:

- Decide on SOAP-RPC or SOAP messaging
- Write or obtain access to a SOAP service
- Write or obtain access to a SOAP client

The first step is to decide if you want to use SOAP for RPC-style calls, in which a remote procedure is invoked on a server, or for messaging, in which a client simply sends pieces of information to a server. I'll detail these processes in the next section. Once you've made that design decision, you need to access, or code up, a service. Of course, since we're all Java pros here, this chapter shows you how to code up your own. Finally, you need to write the client for this service, and watch things take off.

RPC or Messaging?

Your first task is actually not code-related but design-related. You need to determine if you want an RPC service or a messaging one. The first, RPC, is something you should be pretty familiar with after the last chapter. A client invokes a remote procedure on a server somewhere, and then gets some sort of response. In this scenario, SOAP is simply acting as a more extensible XML-RPC system, allowing better error handling and passing of complex types across the network. This is a concept you should already understand, and because it turns out that RPC systems are simple to write in SOAP, I'll start off there. This chapter describes how to write an RPC service, and then an RPC client, and put the system in action.

The second style of SOAP processing is message-based. Instead of invoking remote procedures, it provides for transfer of information. As you can imagine, this is pretty powerful, and doesn't depend on a client knowing about a particular method on some server. It also models distributed systems more closely, allowing packets of data (packet in the figurative sense, not in the network sense) to be passed around, keeping various systems aware of what other systems are doing. It is also more complicated than the simpler RPC-style programming, so I'll cover it in the next chapter with other business-to-business details after you're well grounded in SOAP-RPC programming.

Like most design issues, the actual process of making this decision is left up to you. Look at your application and determine exactly what you want SOAP to do for you. If you have a server and a set of clients that just need to perform tasks remotely, then RPC is probably well suited for your needs. However, in larger systems that are exchanging data rather than performing specific business functions on request, SOAP's messaging capabilities may be a better match.

An RPC Service

With the formalities out of the way, it's time to get going, fast and furious. As you'll recall from the last chapter, in RPC you need a class that is going to have its methods invoked remotely.

Code artifacts

I'll start by showing you some *code artifacts* to have available on the server. These artifacts are classes with methods that are exposed to RPC clients.[2] Rather than use the simple class from last chapter, I offer a slightly more complex example to show you what SOAP can do. In that vein, Example 12-4 is a class that stores a CD inventory, such as an application for an online music store might use. I'm introducing a basic version here, and will add to it later in the chapter.

Example 12-4: The CDCatalog class

```
package javaxml2;

import java.util.Hashtable;

public class CDCatalog {

    /** The CDs, by title */
    private Hashtable catalog;
```

```
    public CDCatalog(  ) {
        catalog = new Hashtable(  );

        // Seed the catalog
        catalog.put("Nickel Creek", "Nickel Creek");
        catalog.put("Let it Fall", "Sean Watkins");
        catalog.put("Aerial Boundaries", "Michael Hedges");
        catalog.put("Taproot", "Michael Hedges");
    }

    public void addCD(String title, String artist) {
        if ((title == null) || (artist == null)) {
            throw new IllegalArgumentException("Title and artist cannot be null
        }
        catalog.put(title, artist);
    }

    public String getArtist(String title) {
        if (title == null) {
            throw new IllegalArgumentException("Title cannot be null.");
        }

        // Return the requested CD
        return (String)catalog.get(title);
    }

    public Hashtable list(  ) {
        return catalog;
    }
}
```

This allows for adding a new CD, searching for an artist by a CD title, and getting all current CDs. Take note that the `list( )` method returns a `Hashtable`, and there is nothing special I have to do to make that work; Apache SOAP provides automatic mapping of the `Hashtable` Java type, much as XML-RPC did.

Compile this class, and make sure you've got everything typed in (or downloaded, if you choose) correctly. Notice that the `CDCatalog` class has no knowledge about SOAP. This means you can take your existing Java classes and expose them through SOAP-RPC, which reduces the work required on your end to move to a SOAP-based architecture if needed.

Deployment descriptors

With the Java coding done, you now need to define a deployment descriptor. This specifies several key things to a SOAP server:

- The URN of the SOAP service for clients to access
- The method or methods available to clients
- The serialization and deserialization handlers for any custom classes

The first is similar to a URL, and required for a client to connect to any SOAP server. The second is exactly what you expect: a list of methods letting the client know what are allowable artifacts for a SOAP client. It also lets the SOAP server, which I'll cover in a

moment, know what requests to accept. The third is a means of telling the SOAP server how to handle any custom parameters; I'll come back to this in the next section when I add some more complex behavior to the catalog.

I'll show you the deployment descriptor and detail each item within it. Example 12-5 is the deployment descriptor for the `CDCatalog` service we're creating.

Example 12-5: The CDCatalog deployment descriptor

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:cd-catalog"
>
  <isd:provider type="java"
                scope="Application"
                methods="addCD getArtist list"
  >
    <isd:java class="javaxml2.CDCatalog" static="false" />
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener
</isd:service>
```

First, I referenced the Apache SOAP deployment namespace, and then supplied a URN for my service through the `id` attribute. This should be something unique across services, and descriptive of the service. I showed about as much originality in naming the service as Dave Matthews did with his band, but it gets the job done. Then, I specified through the `java` element the class to expose, including its package name (through the `class` attribute), and indicated that the methods being exposed were not static ones (through the `static` attribute).

Next, I specified a fault listener implementation to use. Apache's SOAP implementation provides two; I used the first, `DOMFaultListener`. This listener returns any exception and fault information through an additional DOM element in the response to the client. I'll get back to this when I look at writing clients, so don't worry too much about it right now. The other fault listener implementation is `org.apache.soap.server.ExceptionFaultListener`. This listener exposes any faults through an additional parameter returned to the client. Since quite a few SOAP-based applications are already going to be working in Java and XML APIs like DOM, it's common to use the `DOMFaultListener` in most cases.

Deploying the service

At this point, you've got a working deployment descriptor and a set of code artifacts to expose, and you can deploy your service. Apache SOAP comes with a utility to do this task, provided you have done the setup work. First, you need a deployment descriptor for your service, which I just talked about. Second, you need to make the classes for your service available to the SOAP server. The best way to do this is to *jar* up the service class from the last section:

```
jar cvf javaxml2.jar javaxml2/CDCatalog.class
```

Take this *jar* file and drop it into your *lib/* directory (or wherever libraries are auto-loaded

for your servlet engine), and restart your servlet engine.

> WARNING: When you do this, you have created a snapshot of your class file. Changing the code in the *CDCatalog.java* file and recompiling it will not cause the servlet engine to pick up the changes. You'll need to re-*jar* the archive and copy it over to your *lib/* directory each time code changes are made to ensure your service is updated. You'll also want to restart your servlet engine to make sure the changes are picked up by the engine as well.

With your service class (or classes) accessible by your SOAP server, you can now deploy the service, using Apache SOAP's `org.apache.soap.server.ServiceManager` utility class:

```
C:\javaxml2\Ch12>java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter deploy xml\CDCatalogDD.xml
```

The first argument is the SOAP server and RPC router servlet, the second is the action to take, and the third is the relevant deployment descriptor. Once this has executed, verify your service was added:

```
(gandalf)/javaxml2/Ch12$ java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter list
Deployed Services:
        urn:cd-catalog
        urn:AddressFetcher
        urn:xml-soap-demo-calculator
```

At a minimum, this should show any and all services you have available on the server. Finally, you can easily undeploy the service, as long as you know its name:

```
C:\javaxml2\Ch12>java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter undeploy urn:cd-catalog
```

Every time you update your service code, you must undeploy and then redeploy to ensure the SOAP server is running the newest copy.

An RPC Client

Next up is the client. I'm going to keep things simple, and just write a couple of command-line programs that invoke SOAP-RPC. It would be impossible to try and guess your business case, so I just focus on the SOAP details and let you work out integration with your existing software. Once you have the business portion of your code working, there are some basic steps you'll take in every SOAP-RPC call:

- Create the SOAP-RPC call
- Set up any type mappings for custom parameters
- Set the URI of the SOAP service to use
- Specify the method to invoke

- Specify the encoding to use
- Add any parameters to the call
- Connect to the SOAP service
- Receive and interpret a response

That may seem like a lot, but most of the operations are one- or two-line method invocations. In other words, talking to a SOAP service is generally a piece of cake. Example 12-6 shows the code for the `CDAdder` class, which allows you to add a new CD to the catalog. Take a look at the code, and then I'll walk you through the juicy bits.

Example 12-6: The CDAdder class

```
package javaxml2;

import java.net.URL;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;

public class CDAdder {

    public void add(URL url, String title, String artist)
        throws SOAPException {

        System.out.println("Adding CD titled '" + title + "' by '" +
            artist + "'");

        // Build the Call object
        Call call = new Call(  );
        call.setTargetObjectURI("urn:cd-catalog");
        call.setMethodName("addCD");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        // Set up parameters
        Vector params = new Vector(  );
        params.addElement(new Parameter("title", String.class, title, null));
        params.addElement(new Parameter("artist", String.class, artist, null));
        call.setParams(params);

        // Invoke the call
        Response response;
        response = call.invoke(url, "");

        if (!response.generatedFault(  )) {
            System.out.println("Successful CD Addition.");
        } else {
            Fault fault = response.getFault(  );
            System.out.println("Error encountered: " + fault.getFaultString(  )
        }
    }
```

```
    public static void main(String[] args) {
        if (args.length != 3) {
            System.out.println("Usage: java javaxml2.CDAdder [SOAP server URL]
                "\"[CD Title]\" \"[Artist Name]\"");
            return;
        }

        try {
            // URL for SOAP server to connect to
            URL url = new URL(args[0]);

            // Get values for new CD
            String title = args[1];
            String artist = args[2];

            // Add the CD
            CDAdder adder = new CDAdder(  );
            adder.add(url, title, artist);
        } catch (Exception e) {
            e.printStackTrace(  );
        }
    }
}
```

This program captures the URL of the SOAP server to connect to, as well as information needed to create and add a new CD to the catalog. Then, in the `add( )` method, the code creates the SOAP `Call` object, on which all the interesting interaction occurs. The target URI of the SOAP service and the method to invoke are set on the call, and both match up to values from the service's deployment descriptor from Example 12-5. Next, the encoding is set, which should always be the constant `Constants.NS_URI_SOAP_ENC` unless you have very unique encoding needs.

The program creates a new `Vector` populated with SOAP `Parameter` objects. Each of these represents a parameter to the specified method, and since the `addCD( )` method takes two `String` values, this is pretty simple. Supply the name of the parameter (for use in the XML and debugging), the class for the parameter, and the value. The fourth argument is an optional encoding, if a single parameter needs a special encoding. For no special treatment, the value `null` suffices. The resulting `Vector` is then added to the `Call` object.

Once your call is set up, use the `invoke( )` method on that object. The return value from this method is an `org.apache.soap.Response` instance, which is queried for any problems that resulted. This is fairly self-explanatory, so I'll leave it to you to walk through the code. Once you've compiled your client and followed the instructions earlier in this chapter for setting up your classpath, run the example as follows:

```
C:\javaxml2\build>java javaxml2.CDAdder
   http://localhost:8080/soap/servlet/rpcrouter
   "Riding the Midnight Train" "Doc Watson"

Adding CD titled 'Riding the Midnight Train' by 'Doc Watson'
Successful CD Addition
```

Example 12-7 is another simple class, `CDLister`, which lists all current CDs in the catalog. I won't go into detail on it, as it's very similar to Example 12-6, and is mainly a

reinforcement of what I've already talked about.

Example 12-7: The CDLister class

```
package javaxml2;

import java.net.URL;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;

public class CDLister {

    public void list(URL url) throws SOAPException {
        System.out.println("Listing current CD catalog.");

        // Build the Call object
        Call call = new Call(  );
        call.setTargetObjectURI("urn:cd-catalog");
        call.setMethodName("list");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        // No parameters needed

        // Invoke the call
        Response response;
        response = call.invoke(url, "");

        if (!response.generatedFault(  )) {
            Parameter returnValue = response.getReturnValue(  );
            Hashtable catalog = (Hashtable)returnValue.getValue(  );
            Enumeration e = catalog.keys(  );
            while (e.hasMoreElements(  )) {
                String title = (String)e.nextElement(  );
                String artist = (String)catalog.get(title);
                System.out.println("  '" + title + "' by " + artist);
            }
        } else {
            Fault fault = response.getFault(  );
            System.out.println("Error encountered: " + fault.getFaultString(  )
        }
    }

    public static void main(String[] args) {
        if (args.length != 1) {
            System.out.println("Usage: java javaxml2.CDAdder [SOAP server URL]"
            return;
        }

        try {
            // URL for SOAP server to connect to
            URL url = new URL(args[0]);
```

```
            // List the current CDs
            CDLister lister = new CDLister(  );
            lister.list(url);
        } catch (Exception e) {
            e.printStackTrace(  );
        }
    }
}
```

The only difference in this method from the `CDAdder` class is that the `Response` object has a return value (the `Hashtable` from the `list( )` method). This is returned as a `Parameter` object, which allows a client to check its encoding and then extract the actual method return value. Once that's done, the client can use the returned value like any other Java object, and in the example simply runs through the CD catalog and prints out each one. You can now run this additional client to see it in action:

```
C:\javaxml2\build>java javaxml2.CDLister
    http://localhost:8080/soap/servlet/rpcrouter
Listing current CD catalog.
  'Riding the Midnight Train' by Doc Watson
  'Taproot' by Michael Hedges
  'Nickel Creek' by Nickel Creek
  'Let it Fall' by Sean Watkins
  'Aerial Boundaries' by Michael Hedges
```

That's really all there is to basic RPC functionality in SOAP. I'd like to push on a bit, though, and talk about a few more complex topics.

Going Further

Although you can now do everything in SOAP you knew how to do in XML-RPC, there is a lot more to SOAP. As I said in the beginning of the chapter, two important things that SOAP brings to the table are the ability to use custom parameters with a minimal amount of effort, and more advanced fault handling. In this section, I cover both of these topics.

Custom Parameter Types

The most limiting thing with the CD catalog, at least at this point, is that it stores only the title and artist for a given CD. It is much more realistic to have an object (or set of objects) that represents a CD with the title, artist, label, track listings, perhaps a genre, and all sorts of other information. I'm not going to build this entire structure, but will move from a title and artist to a `CD` object with a title, artist, and label. This object needs to be passed from the client to the server and back, and demonstrates how SOAP can handle these custom types. Example 12-8 shows this new class.

Example 12-8: The CD class

```
package javaxml2;

public class CD {

    /** The title of the CD */
    private String title;
```

```
    /** The artist performing on the CD */
    private String artist;

    /** The label of the CD */
    private String label;

    public CD(  ) {
        // Default constructor
    }

    public CD(String title, String artist, String label) {
        this.title = title;
        this.artist = artist;
        this.label = label;
    }

    public String getTitle(  ) {
        return title;
    }

    public void setTitle(String title) {
        this.title = title;
    }

    public String getArtist(  ) {
        return artist;
    }

    public void setArtist(String artist) {
        this.artist = artist;
    }

    public String getLabel(  ) {
        return label;
    }

    public void setLabel(String label) {
        this.label = label;
    }

    public String toString(  ) {
        return "'" + title + "' by " + artist + ", on " +
            label;
    }
}
```

This requires a whole slew of changes to the `CDCatalog` class as well. Example 12-9 shows a modified version of this class with the changes that use the new `CD` support class highlighted.

Example 12-9: An updated CDCatalog class

```
package javaxml2;

import java.util.Hashtable;

public class CDCatalog {
```

```
    /** The CDs, by title */
    private Hashtable catalog;

    public CDCatalog(  ) {
        catalog = new Hashtable(  );

        // Seed the catalog
        addCD(new CD("Nickel Creek", "Nickel Creek", "Sugar Hill"));
        addCD(new CD("Let it Fall", "Sean Watkins", "Sugar Hill"));
        addCD(new CD("Aerial Boundaries", "Michael Hedges", "Windham Hill"));
        addCD(new CD("Taproot", "Michael Hedges", "Windham Hill"));
    }

    public void addCD(CD cd) {
        if (cd == null) {
            throw new IllegalArgumentException("The CD object cannot be null.")
        }
        catalog.put(cd.getTitle(  ), cd);
    }

    public CD getCD(String title) {
        if (title == null) {
            throw new IllegalArgumentException("Title cannot be null.");
        }

        // Return the requested CD
        return (CD)catalog.get(title);
    }

    public Hashtable list(  ) {
        return catalog;
    }
}
```

In addition to the obvious changes, I've also updated the old `getArtist(String title)` method to `getCD(String title)`, and made the return value a `CD` object. This means the SOAP server will need to serialize and deserialize this new class, and the client will be updated. First, I look at an updated deployment descriptor that details the serialization issues related to this custom type. Add the following lines to the deployment descriptor for the CD catalog, as well as changing the available method names to match the updated `CDCatalog` class:

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
             id="urn:cd-catalog"
>
  <isd:provider type="java"
                scope="Application"
                methods="addCD getCD list"
  >
    <isd:java class="javaxml2.CDCatalog" static="false" />
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener

  <isd:mappings>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:x="urn:cd-catalog-demo" qname="x:cd"
```

```
            javaType="javaxml2.CD"
            java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer
            xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer
  </isd:mappings>
</isd:service>
```

The new element, `mappings`, specifies how a SOAP server should handle custom parameters such as the `CD` class. First, define a `map` element for each custom parameter type. For the `encodingStyle` attribute, at least as of Apache SOAP 2.2, you should always supply the value http://schemas.xmlsoap.org/soap/encoding/, the only encoding currently supported. You need to supply a namespace for the custom type and then the name of the class, with this namespace prefix, for the type. In my case, I used a "dummy" namespace and the simple prefix "x" for this purpose. Then, using the `javaType` attribute, supply the actual Java class name: `javaxml2.CD` in this case. Finally, the magic occurs in the `java2XMLClassName` and `xml2JavaClassName` attributes. These specify a class to convert from Java to XML and from XML to Java, respectively. I've used the incredibly handy `BeanSerializer` class, also provided with Apache SOAP. If your custom parameter is in a JavaBean format, this serializer and deserializer will save you from having to write your own. You need to have a class with a default constructor (remember that I defined an empty, no-args constructor within the `CD` class), and expose all the data in that class through `setXXX` and `getXXX` style methods. Since the `CD` class fits the bill here, the `BeanSerializer` works perfectly.

> NOTE: It's no accident that the `CD` class follows the JavaBean conventions. Most data classes fit easily into this format, and I knew I wanted to avoid writing my own custom serializer and deserializer. These are a pain to write (not overly difficult, but easy to mess up), and I recommend you go to great lengths to try and use the Bean conventions in your own custom parameters. In many cases, the Bean conventions only require that a default constructor (with no arguments) is present in your class.

Now recreate your service *jar* file. Then, redeploy your service:

```
(gandalf)/javaxml2/Ch12$ java org.apache.soap.server.ServiceManagerClient
    http://localhost:8080/soap/servlet/rpcrouter xml/CDCatalogDD.xml
```

> WARNING: If you have kept your servlet engine running and the service deployed all this time, you'll need to restart the servlet engine to activate the new classes for the SOAP service, and redeploy the service.

At this point, all that's left is modifying the client to use the new class and methods. Example 12-10 is an updated version of the client class `CDAdder`. The changes from the previous version of the class are highlighted.

Example 12-10: The updated CDAdder class

```
package javaxml2;

import java.net.URL;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
```

```
import org.apache.soap.SOAPException;
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.encoding.soapenc.BeanSerializer;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;
import org.apache.soap.util.xml.QName;

public class CDAdder {

    public void add(URL url, String title, String artist, String label)
        throws SOAPException {

        System.out.println("Adding CD titled '" + title + "' by '" +
            artist + "', on the label " + label);

        CD cd = new CD(title, artist, label);

        // Map this type so SOAP can use it
        SOAPMappingRegistry registry = new SOAPMappingRegistry(  );
        BeanSerializer serializer = new BeanSerializer(  );
        registry.mapTypes(Constants.NS_URI_SOAP_ENC,
            new QName("urn:cd-catalog-demo", "cd"),
            CD.class, serializer, serializer);

        // Build the Call object
        Call call = new Call(  );
        call.setSOAPMappingRegistry(registry);
        call.setTargetObjectURI("urn:cd-catalog");
        call.setMethodName("addCD");
        call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

        // Set up parameters
        Vector params = new Vector(  );
        params.addElement(new Parameter("cd", CD.class, cd, null));
        call.setParams(params);

        // Invoke the call
        Response response;
        response = call.invoke(url, "");

        if (!response.generatedFault(  )) {
            System.out.println("Successful CD Addition.");
        } else {
            Fault fault = response.getFault(  );
            System.out.println("Error encountered: " + fault.getFaultString(  )
        }
    }

    public static void main(String[] args) {
        if (args.length != 4) {
            System.out.println("Usage: java javaxml2.CDAdder [SOAP server URL]
                "\"[CD Title]\" \"[Artist Name]\" \"[CD Label]\"");
            return;
        }

        try {
            // URL for SOAP server to connect to
            URL url = new URL(args[0]);
```

```
            // Get values for new CD
            String title = args[1];
            String artist = args[2];
            String label = args[3];

            // Add the CD
            CDAdder adder = new CDAdder( );
            adder.add(url, title, artist, label);
        } catch (Exception e) {
            e.printStackTrace( );
        }
    }
}
```

The only really interesting change is in dealing with the mapping of the `CD` class:

```
        // Map this type so SOAP can use it
        SOAPMappingRegistry registry = new SOAPMappingRegistry( );
        BeanSerializer serializer = new BeanSerializer( );
        registry.mapTypes(Constants.NS_URI_SOAP_ENC,
            new QName("urn:cd-catalog-demo", "cd"),
            CD.class, serializer, serializer);
```

This is how a custom parameter can be encoded and sent across the wire. I already discussed how the `BeanSerializer` class could be used to handle parameters in the JavaBean format, such as the `CD` class. To specify that to the server, I used the deployment descriptor; however, now I need to let the client know to use this serializer and deserializer. This is what the `SOAPMappingRegistry` class allows. The `mapTypes( )` method takes in an encoding string (again, using the constant `NS_URI_SOAP_ENC` is the best idea here), and information about the parameter type a special serialization should be used for. First, a `QName` is supplied. This is why the odd namespacing was used back in the deployment descriptor; you need to specify the same URN here, as well as the local name of the element (in this case "CD"), then the Java `Class` object of the class to be serialized (`CD.class`), and finally the class instance for serialization and deserialization. In the case of the `BeanSerializer`, the same instance works for both. Once all this is set up in the registry, let the `Call` object know about it through the `setSOAPMapping-Registry( )` method.

You can run this class just as before, adding the CD label, and things should work smoothly:

```
C:\javaxml2\build>java javaxml2.CDAdder
    http://localhost:8080/soap/servlet/rpcrouter
    "Tony Rice" "Manzanita" "Sugar Hill"
Adding CD titled 'Tony Rice' by 'Manzanita', on the label Sugar Hill
Successful CD Addition.
```

I'll leave it up to you to modify the `CDLister` class in the same fashion, and the downloadable samples have this updated class as well.

> NOTE: You might think that since the `CDLister` class doesn't deal directly with a `CD` object (the return value of the `list( )` method was a `Hashtable`), you don't need to make any changes. However, the returned `Hashtable` contains instances of CD objects. If SOAP doesn't know how to deserialize these, your client is going to give you an error. Therefore, you must specify a

`SOAPMappingRegistry` instance on the `Call` object to make things work.

Better Error Handling

Now that you're tossing around custom objects, making RPC calls, and generally showing up everyone else in the office, let me talk about a less exciting topic: error handling. In any network transaction, many things can go wrong. The service isn't running, an error occurs on the server, objects can't be found, classes are missing, and a whole lot of other problems can arise. Until now, I just used the `fault.getString( )` method to report errors. But this method isn't always very helpful. To see it in action, comment out the following line in the `CDCatalog` constructor:

```
    public CDCatalog(  ) {
        //catalog = new Hashtable(  );

        // Seed the catalog
        addCD(new CD("Nickel Creek", "Nickel Creek", "Sugar Hill"));
        addCD(new CD("Let it Fall", "Sean Watkins", "Sugar Hill"));
        addCD(new CD("Aerial Boundaries", "Michael Hedges", "Windham Hill"));
        addCD(new CD("Taproot", "Michael Hedges", "Windham Hill"));
    }
```

Recompile, restart your server engine, and redeploy. The result is that a `NullPointerException` occurs when the class constructor tries to add a CD to an uninitialized `Hashtable`. Running the client will let you know an error has occurred, but not in a very meaningful way:

```
(gandalf)/javaxml2/build$ java javaxml2.CDLister
    http://localhost:8080/soap/servlet/rpcrouter
Listing current CD catalog.
Error encountered: Unable to resolve target object: null
```

This isn't exactly the type of information you need to track down the problem. However, the framework is in place to do a better job of error handling; remember the `DOMFaultListener` you specified as the value of the `faultListener` element? This is where it comes into play. The returned `Fault` object in the case of a problem (as in this one) contains a DOM `org.w3c.dom.Element` with detailed error information. First, add an import statement for `java.util.Iterator` to your client source code:

```
import java.net.URL;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Iterator;
import java.util.Vector;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.SOAPException;
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.encoding.soapenc.BeanSerializer;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;
import org.apache.soap.util.xml.QName;
```

Next, make the following change to how errors are handled in the `list( )` method:

```
        if (!response.generatedFault( )) {
            Parameter returnValue = response.getReturnValue( );
            Hashtable catalog = (Hashtable)returnValue.getValue( );
            Enumeration e = catalog.keys( );
            while (e.hasMoreElements( )) {
                String title = (String)e.nextElement( );
                CD cd = (CD)catalog.get(title);
                System.out.println("  '" + cd.getTitle( ) + "' by " +
cd.getArtist( ) +
                    " on the label " + cd.getLabel( ));
            }
        } else {
            Fault fault = response.getFault( );
            System.out.println("Error encountered: " + fault.getFaultString( )

            Vector entries = fault.getDetailEntries( );
            for (Iterator i = entries.iterator(); i.hasNext( ); ) {
                org.w3c.dom.Element entry = (org.w3c.dom.Element)i.next( );
                System.out.println(entry.getFirstChild().getNodeValue( ));
            }
        }
```

By using the `getDetailEntries( )` method, you get access to the raw data supplied by the SOAP service and server about the problem. The code iterates through these (there is generally only a single element, but it pays to be careful) and grabs the DOM `Element` contained within each entry. Essentially, here's the XML you are working through:

```
<SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Server.BadTargetObjectURI</faultcode>
  <faultstring>Unable to resolve target object: null</faultstring>
  <stacktrace>Here's what we want!</stackTrace>
</SOAP-ENV:Fault>
```

In other words, the `Fault` object gives you access to the portion of the SOAP envelope that deals with errors. Additionally, Apache SOAP provides a Java stack trace if errors occur, and that provides the detailed information needed to troubleshoot problems. By grabbing the `stackTrace` element and printing the `Text` node's value from that `Element`, your client will now print out the stack trace from the server. Compile these changes and rerun the client. You should get the following output:

```
C:\javaxml2\build>java javaxml2.CDLister http://localhost:8080/soap/servlet/rpc
outer
Listing current CD catalog.
Error encountered: Unable to resolve target object: null
java.lang.NullPointerException
        at javaxml2.CDCatalog.addCD(CDCatalog.java:24)
        at javaxml2.CDCatalog.<init>(CDCatalog.java:14)
        at java.lang.Class.newInstance0(Native Method)
        at java.lang.Class.newInstance(Class.java:237)
```

This goes on for a bit, but you can see the juicy bits of information indicating that a `NullPointerException` occurred, and even get the line numbers on the server classes where the problems happened. The result of this fairly minor change is a much more robust means of handling errors. That should prepare you for tracking down bugs on your server

classes. Oh, and be sure to change your `CDCatalog` class back to a version that won't cause these errors before moving on!

What's Next?

The next chapter is a direct continuation of these topics. More than ever, XML is becoming the cornerstone of business-to-business activity, and SOAP is key to that. In the next chapter, I'll introduce two important technologies to you, UDDI and WSDL. If you have no idea what those are, you're in the right place. You'll learn how they all fit together to form the backbone of web services architectures. Get ready to finally find out what the web services, peer-to-peer craze is all about.

---

1. There's a lot of talk about running SOAP over other protocols, like SMTP (or even Jabber). This isn't part of the SOAP standard, but it may be added in the future. Don't be surprised if you see it discussed.

2. You can use scripts through the Bean Scripting Framework, but for the sake of space I won't cover that here. Check out the upcoming O'Reilly SOAP book, as well as the online documentation at http://xml.apache.org/soap, for more details on script support in SOAP.

Back to: Java & XML, 2nd Edition
